7-21-1987

Interactive Virtual Debugger for GPSS/H

Eugene Johnson

Rochester Institute of Technology
School of Computer Science & Technology

INTERACTIVE VISUAL DEBUGGER FOR GPSS/H

by

Eugene Johnson

A thesis, submitted to the
Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

Guy Johnson

Professor Guy Johnson

Peter G. Anderson

Professor Peter Anderson

Chris Comte

Professor Chris Comte

July 21, 1987

Statement for Granting Permission to Reproduce

Thesis Title: Interactive Visual Debugger for GPSS/H

I, Eugene Johnson, hereby grant permission to the Wallece Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: 08/04/87

Eugene Johnson

ABSTRACT

GPSS (General Purpose System Simulator) is a language, designed to aid the computer modeling and simulation of a wide variety of different real life systems. As with any other large programming project, debugging GPSS programs is unavoidable and often difficult.

The present thesis describes an Interactive Visual Debugging System for GPSS/H which attempts to simplify the debugging task by allowing the programmer to observe the actual behavior of the model in simulated real time, while preserving all traditional interactive debugging tools - breakpoints, system traps, selective displays, etc.

The present version of the Interactive Visual Debugger is developed for the UNIX operating system and is written in the 'C' programming language. The system can be readily used on all terminals capable of running the UNIX 'curses' library package. Because of its modular design, the system can be modified to accommodate additional terminal types, or to run under different operating systems.

COMPUTING REVIEWS SUBJECT CODES

Primary Code:	D.2.5.	Software Engineering. Testing and Debugging.
Secondary Codes:	I.3.4.	Computer Graphics. Graphics Utilities.
	I.6.2.	Simulation and Modeling. Simulation Languages.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. General Remarks

Simulation is the technique of constructing and running a model of a real system in order to study the behavior of that system, without disrupting the environment of the real system [3]. It is the tool that is used when a problem can not be solved simply and satisfactorily by analytical methods. Although currently there is no scientific theory to guarantee the validity of a simulation process before the actual experiment is performed, when properly used simulation can provide valuable insight on how the components of a system are going to behave.

In any computer simulation there are two distinct stages: first, systems analysis - to define and build a model of the system; and, second, computer programming - to code the model for data processing. One attempt to simplify this process is illustrated by the development of the simulation language called General Purpose System Simulator (GPSS).

GPSS was not designed for any particular class of systems. Its goal was to define a certain basic set of capabilities, which are characteristic of a wide variety of different systems. The language can be used as easily for modeling the scheduling algorithm of a CPU as it can be used for determining the utilization of the cashiers in a grocery store.

1.2. History and Evolution of GPSS

GPSS was developed by Geoffrey Gordon, an IBM employee, and presented in two papers: "A General Purpose System Simulator" [6] and "A General Purpose Digital Simulator and Examples of its Application: Part 1 - Description of the Simulator" [7].

The first version of GPSS defined a set of 25 specific blocks. Each block represented a basic system action and had an associated time to perform this action. The modeled system

was described in terms of a combination of blocks that could be used repeatedly. General units of traffic in the modeled system (such as vehicles, people, goods, processes, etc.) were represented by transactions and these were the dynamic components of the system. The transactions moved through the block diagram under the control of the blocks and were created and destroyed as required. During this process, different statistics were collected and used to evaluate the performance of the system and to draw appropriate conclusions.

The first major changes in the design of the language were made in GPSS III, released in 1965. Some of these changes were:

- elimination of the time delays associated with every block and introduction of a single ADVANCE block to model all passages of time.
- removal of selection factors from all blocks and their replacement with the single TRANSFER block.
- introduction of user chains in addition to the standard current events and future events chains.
- addition of new blocks (DEPART, SPLIT, GATHER).
- increasing the power of the HELP block.

As a result of these and other improvements, the efficiency of execution and power of the language were greatly improved.

Some other versions of GPSS since 1965 are GPSS/360, GPSS V, GPSS/C, etc. In all cases the basic structure of the language remained unaltered since GPSS III and all further releases are upward-compatible. The changes were concerned with enhancing the power of the language, relaxing the syntax, introducing debugging tools, improving the speed of execution, etc.

The version of the language we are interested in is GPSS/H. It offers excellent performance characteristics and many extensions beyond previous implementations. The principal advantages are [9]:

- Greatly improved execution speeds. Typical models run 4-5 times as fast under GPSS/H as under GPSS V. This is due primarily to the effectiveness of the GPSS/H compiler.

- Design for interactive use and a simple, but powerful command language for setting breakpoints, stepping through the model, selectively displaying model data, etc.
- Many extensions beyond the "standard" GPSS, including I/O statements, dynamic control statement logic (IF-THEN-ELSE, DO, GOTO), and greater flexibility in the rules for coding statements.

1.3. Development of GPSS Models

The development of a GPSS model usually goes through the following stages [9]:

(1) Choosing an appropriate level of abstraction. If a model of a system is too abstract, the results may be insufficient to draw proper conclusions. On the other hand, if a model is too detailed, completion may take too long and the costs of development may become to high for the model to be of value.

(2) Mapping the elements of the system into GPSS entities and defining additional entities for performing computations and collecting statistics. GPSS provides three major entity classes:

- Equipment entities: facilities, storages, logic switches.
- Computational entities: arithmetic and boolean variables, functions, etc.
- Statistical entities: queues, tables.

(3) Specifying the rules by which the particular system operates. The logic of a model is described by using "blocks", selected from a collection of different block types. (In GPSS/H there are over 50 block types.) The construction of a model from blocks is usually done by flowcharting the model as a GPSS Block diagram, drawing each GPSS block with a characteristic shape.

(4) Coding, compiling and debugging of the GPSS model.

Appendix A gives an example of a simple GPSS model together with the results from the simulation. It is intended for the unfamiliar with the GPSS language reader and attempts to illustrate the above points.

1.4. Block Diagrams and Transactions

The most obvious characteristic of a GPSS Block diagram is the use of many distinctively shaped blocks. Such a diagram provides an extremely convenient notation for expressing the rules by which the system being modeled operates. Since the rules usually do not change during the course of simulation, the Block diagram may be considered to be the static element of a program.

The dynamic elements of the GPSS program which traverse the Block diagram are called *transactions*. The meaning of a transaction is determined by the modeler. At any given time there can be many transactions "in process" at various points of the Block diagram. It must be noted that, depending on the system modeled, it is possible for the GPSS Block diagram to have disjoint segments.

1.5. Program Verification

Up to this moment the discussion was primarily focused on the GPSS language. In this section a few general remarks will be made on program debugging.

Once a computer program has been developed, it has to be tested, or verified, so that eventual errors can be corrected. This process is commonly referred to as **debugging**. It is very well known that this part of the program developing cycle consumes a lot of time and programming resources. This is the reason why throughout the years many efforts have been made to facilitate the debugging process as much as possible. These efforts are focused in many different directions that can be loosely divided into two major groups:

- Error prevention, and
- Error detection.

The efforts to prevent errors from occurring in the first place, resulted in the development of a wide variety of different practices. Among these are:

- the emergence of structured and modular hierarchical programming,
- the development of scientific software engineering management practices,
- the development of strongly typed programming languages,
- the development of language sensitive editors, etc.

The efforts to facilitate the discovery of errors (whether syntactical or logical) resulted, among other things, in:

- development of languages with better syntax diagnostics.
- development of language verifiers (for example the *lint* program).
- including of special debugging statements in some of the languages (for example the DISPLAY and WITH DEBUGGING statements in COBOL).
- development of dump and trace formatting facilities (for example the VM/CMS operating system).
- development of general and special purpose debuggers.

For the purposes of this thesis we are interested in the development of debuggers. Practically every operating system environment provides a debugger in one form or another. In the case of the UNIX system we have either *adb* or *sdb*, in the case of the IBM's VM/SP it is a set of control program (CP) commands (ADSTOP, PER, DISPLAY, etc.), in the case of the VAX/VMS operating system - the VAX/VMS Symbolic Debugger. The above are all examples of general purpose debuggers.

Almost all debuggers provide capabilities for memory dumps, register and PSW display, selective examination and altering of memory locations, controlled step by step execution of the program, imbedding of trace and break points, etc. Unfortunately, learning to use these

features is often not a trivial task. Deliberate efforts were made throughout the last two decades to make the debuggers more "user friendly". The following is a list of some of the features of the VAX/VMS Symbolic Debugger which is considered to be representative of the latest trends in debugger development:

- The VAX/VMS Debugger is symbolic. This allows the user to refer to memory locations by their symbolic names or as offsets from symbolic names and not by their absolute or virtual addresses.

- The VAX/VMS Symbolic Debugger is multi-language. It supports all VAX "native" languages; i.e., during a single session one can debug a program written in any number of languages. For languages that are not supported, the Debugger provides an environment which accepts commonly used operators and data types.

- The Debugger can display the source code of the module being debugged. It can also convert the object code into pseudo-assembler, which allows the user to see the assembly language equivalent of the high level language statements being executed.

- Virtual addresses (including register and stack locations) can be translated into symbols or offsets from symbols.

- Complete structures (arrays, records) can be displayed with a single command.

- The VAX/VMS Symbolic Debugger can be used in screen mode of operation. In screen mode the user can divide the screen into scrollable windows and display simultaneously different kinds of information: source and assembly code, register and stack contents, debugger input and output. The selected displays are automatically updated.

- The syntax of the Debugger commands is consistent with the Digital Command Language (DCL)* syntax and the syntax of the VAX extensions in the different languages.

- Several DCL features - command procedures, log and initialization files, symbol definition - are supported by the Debugger as well.

- The Debugger has a mechanism for keypad and function key definition which allows an easy way of issuing fairly elaborate or frequently used commands.

---

*) DCL is the command language of the VAX/VMS operating system.

Often, large software products will offer specialized built-in debuggers, for example: the built-in debugger of the MACRO-11 simulator MAXIM, the Execution Diagnostic Facility (EDF) in the CICS-VS on-line software monitor, the interactive debuggers in GPSS/C and GPSS/H, etc.

2. PROBLEM STATEMENT

There are two distinguishing features of GPSS that should be pointed out: first, all blocks in GPSS have associated with them distinctive graphical symbols which allow us to describe the modeled system as a connected network of Block diagrams representing the sequence of events. And second, the basic units in GPSS, called "transactions", could vary in nature depending on the particular system to be modeled, but the important thing for us is that they always go through the same cycle: they are created, under the control of different blocks they move through the model (during which time different statistics are collected), and then they are destroyed.

The above features make the conceptual interpretation of almost every GPSS Block diagram possible. This can be used for facilitating the debugging of GPSS programs.

As is the case with all other programming projects, debugging GPSS models can be time consuming, tedious, frustrating and, at the same time, unavoidable. Most modern versions of GPSS (including GPSS/H) provide built-in interactive debugging systems but they are difficult to use, if for no other reason, simply because of their voluminous output which very quickly becomes tiresome to follow.

The prime objective of this thesis is to develop an interactive visual debugging system that will allow the user to observe and control the behavior of the model while the simulation is in progress.

Guided by the ancient Chinese maxim that a picture is worth a thousand words, the proposed system enables the user to really see the movement of the separate transactions through the model. Some of the more important statistics (like clock time, utilization of facilities, queue and storage contents, etc.) are displayed and constantly updated. The user can specify what portion of the model and what set of statistics are to be displayed and, with few minor exceptions, has available the full power and capabilities of the existing GPSS/H debugging system.

In addition to pure debugging, the proposed system will be helpful in the more complete evaluation of the model. The ability to watch the dynamic behavior of the modeled system during the simulation run can result, for example, in the discovery of temporary bottle-neck points or under-utilizations that otherwise might not be reflected in the final report.

3. Functional Specification

From this point on, it is assumed that the resident operating system is UNIX. Certain modifications will be necessary for different operating systems, but keeping in mind that the underlining philosophy of the design fully complies with the principles of structured modular programming, such modifications should be fairly easy to make. It is assumed that the reader is familiar with the UNIX operating system and some of its basic capabilities.

3.1. Functions Performed

The command that invokes the Interactive Visual Debugger is dependent on the resident operating system. For UNIX, the synopsis of the command is:

gpsdbg [options] file[.gps]

The command is fully described in Section 6.1. The options provide information such as: type of display device used, starting point of the Block diagram section to be displayed, delay factor for transaction movements, etc. It should be noted that all options have reasonable default values. This will allow the system to be used (even though not to its fullest capabilities) by people with limited knowledge of the proposed package.

After the system has been invoked, the requested section of the Block diagram is drawn on the screen. If not specified otherwise, the displayed section starts from Block 1 in the source code. The number of blocks displayed depends on the device used but usually is not less than 15.

If a particular terminal type is specified with the -T option and the Block symbols are implemented for the requested terminal, then the Block diagram and the rest of the display is done with high resolution graphics in "graphics mode"*, otherwise, the Block diagram and the

rest of the display is done in "character mode".*

If syntax errors are encountered in the source code, an error message is displayed and the session terminated.

Simultaneously with drawing the Block diagram, a symbol table is created with all necessary information for the subsequent proper functioning of the system.

All activities described so far can be classified as the "static" part of a debugging session. These are things that are all executed once and do not change for the remaining of the session.

The "dynamic" part starts with the automatic invocation of the GPSS/H compiler in debug mode (option TEST). The user commands are read from the standard input, interpreted and passed to the GPSS/H system so that the process can continue. The output from the standard GPSS/H is trapped, interpreted, and as a result, different modules are invoked and appropriate actions taken to display the information on the standard output in a new form.

The proposed system also incorporates a HELP function. When invoked by the user, it will list the most frequently used commands together with a brief description of their function. A detailed description of all commands is given in the User's Manual in Section 6.

3.2. Limitations and Restrictions

The limitations and restrictions of the proposed system can be divided in three major groups: first, hardware imposed; second, restrictions on the original built-in debugger; third,

---

*) In "graphics" mode the display is accessed on a pixel level and the graphics capabilities of the device are used to their fullest potential. In "character" mode the Block diagram and the rest of the information are built from the standard character set for the particular device.

limitations and restrictions in representing the Block diagram.

From the debugger's point of view, user display stations (terminals) can be divided into the following categories:

- High resolution graphics terminals implemented in the proposed system - the Block diagram and the rest of the information from the debugger are displayed in graphics mode.
- High resolution graphics terminals NOT implemented in the proposed system, but capable of running the *curses** package - the Block diagram and the rest of the information from the debugger are displayed in character mode.
- Ordinary terminals capable of running the *curses* package - the Block diagram and the rest of the information from the debugger are displayed in character mode.
- Ordinary terminals NOT capable of running the *curses* package, but implemented in the proposed system - the Block diagram and the rest of the information from the debugger are displayed in either graphics or character mode, depending on the particular device and method of implementation.
- Ordinary graphics terminals NOT capable of running the *curses* package and NOT implemented in the proposed system - the proposed system can not be used with such devices.

The limitations and restrictions imposed on the original built-in debugger are as follows:

- The DX command (used for displaying information in hexadecimal form and intended primarily for systems programmers) is not supported.
- The output of the DISPLAY command is limited to displaying information for only one model statistic at a time. In certain cases the output is limited to the most important information only. The reason for imposing these restrictions is the limited space available on the screen.

The limitations and restrictions imposed in representing the Block diagram are as follows:

---

*) The *curses* package in UNIX is a set of routines that give the user a method of updating screens with reasonable optimization. The terminal must be described in the */etc/termcap* terminal capability data base.

- Certain blocks (like SAVEVALUE, BPUTPIC, PRINT, TRACE, etc.), which are either a block form of control statements or can not refuse entrance to and do not alter the dynamic behavior of a transaction) can not be represented in the Block diagram drawn on the screen. The idea is to save screen space at the expense of "unimportant" blocks. A full list of the unsupported blocks is given in Appendix C.

- Macros are not supported in the current version of the debugger.

3.3. User Inputs

The user invokes the debugger with the *gpsdbg* command (see Section 6.1). If there are syntax or other fatal errors, the system will stop with an appropriate message either during the phase of building the Block diagram or later, during compilation or execution time.

Provided there are no fatal errors, the user is expected to type in appropriate commands from the set described in Section 6. Unsupported commands result in either error messages or are simply ignored by the system. The goal, again, has been to make the system as "user friendly" as possible.

3.4. User Outputs

The output of the system can be divided into two types:

- output displayed on the terminal, and

- output written into a file.

On the screen the user sees the requested section of the Block diagram, the movements of different transactions (represented by a blinking asterisk or transaction number) and a variety of model statistics. There is space left for error messages, HELP instructions, etc. The actual layout varies, depending on the particular terminal. Sample screen layouts are shown in Appendix B.

The original GPSS/H system creates a disk file in which the source code and the result from the execution are recorded. This output is often referred to as bulk (standard) output. The name of this file is the same as the name of the file containing the source code, but with *.log* substituting the required suffix *.gps*.

Using the SET TLOG=ON and SET TLOG=OFF commands, the user can control the writing into the standard .log file a log of the interactive input and output from the debugging session. When TLOG is ON, all input and output lines are written into the bulk output file. The input lines are prefixed with an arrow.

4. ARCHITECTURAL DESIGN

During the design and development of the present project, a deliberate attempt was made to conform as much as possible to the established modern software engineering practices [14], [15]. The process went through the following main stages:

- problem definition
- initial high level design
- development of a limited version of the system and its testing
- enhancing the high level design
- development of new features
- system testing
- preparation of program documentation
- preparation of user manual

The fundamental principle upon which the design is based is that the system should be made as easy to use as possible. This includes keeping the syntax of the invoking command similar to the syntax of the rest of the commands of the host operating system, keeping the syntax of the Interactive Debugger commands as close as possible to the syntax of the original GPSS/H debugger commands, attempting to cover for the lack of experience of the user by providing reasonable default settings and actions, etc.

A secondary, but still very important goal was building the system in such a way as to assure easy modification and maintenance.

In accordance with the principles of the Top-Down structured design philosophy, the whole system is broken into two major segments:

A. Block diagram building modules, and

B. Actual debugging modules.

The above division is arrived to easily because of the nature of the problem.

The implementation of the first segment is relatively straight-forward and is discussed in more detail in Section 7.2.

The debugging part presented some problems, primarily because the source code for the GPSS/H compiler was not available. It was decided to divide this section into two modules, positioned like filters before and after the GPSS/H processor. In this manner the actual GPSS/H processor remains intact and can be used in "normal" debug mode without any further complications.

5. SYSTEM SPECIFICATIONS

This section is primarily focused on the data flow in the GPSS/H Visual Debugging System in relation to the data flow of the original GPSS/H processor. The system's module hierarchy and inter-module interfaces are described in Section 7.

5.1. Standard GPSS/H I/O Architecture

The standard GPSS/H I/O architecture is depicted in Figure 1.

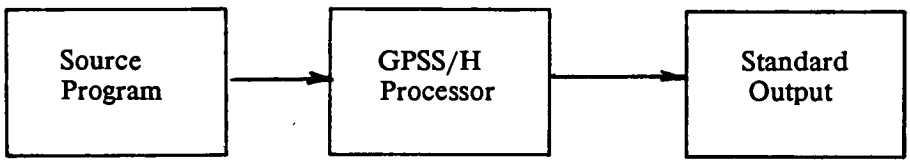


Fig. 1. Standard GPSS/H I/O architecture.

Typically, but not necessarily so, the source program comes from a disk file and the standard (bulk) output* is directed to a disk file. When the built-in debugger is invoked, the GPSS/H module expects interactive command input and produces interactive output. The I/O architecture can be depicted as shown on Figure 2.

*) The standard output consists of compiler listing and statistical report generated as a result of the execution of the model - see Appendix A for an example.

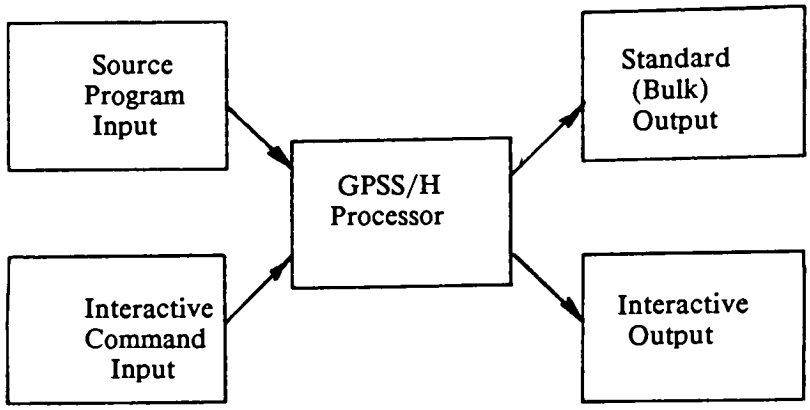


Fig. 2. Standard GPSS/H I/O architecture in Debugging mode.

By default the standard output and the interactive output are separated and the interactive output is directed to the user's terminal. (Although, with the help of the TLOG command, GPSS/H provides means for recording the interactive activities in the standard output as well.)

5.2. Visual Debugger I/O Architecture

The full screen version of the debugger introduces two new major interfaces. These are the command interpreter, positioned between the interactive user command input and the GPSS/H processor, and the interactive output interpreter, positioned between the GPSS/H processor and the output device which, in this case, must be a screen terminal. The interactive command interpreter on its part is in constant communication with the Display Routines Library. The new configuration is shown in Figure 3.

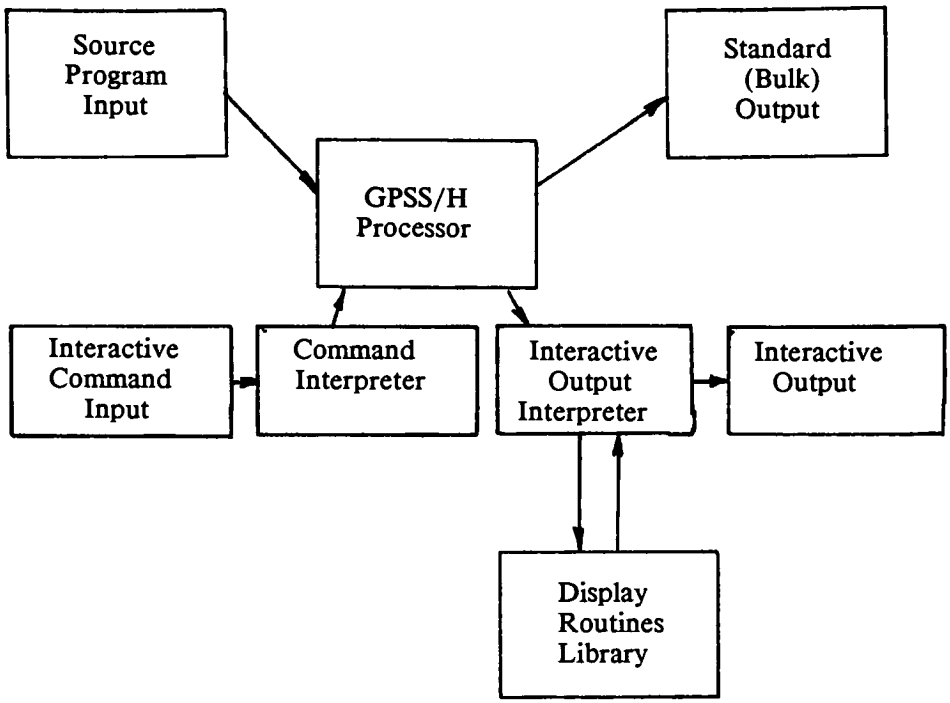


Fig. 3. GPSS/H Visual Debugger I/O architecture (simplified).

The diagram on Figure 3 does not show the modules and interfaces of the Visual Debugger's preprocessor, i.e. the invoking command interpreter, the Block diagram building module and the Graphic Symbols Library. The main role of these modules is to assure the proper invocation of the main GPSS/H processor and the building of the model's Block diagram in accordance with the specified by the user input parameters or system defaults. Schematically, the positioning of the different modules that comprise the Visual Debugger in relation to each other and in relation the user inputs and system outputs, can be depicted as shown in Figure 4.

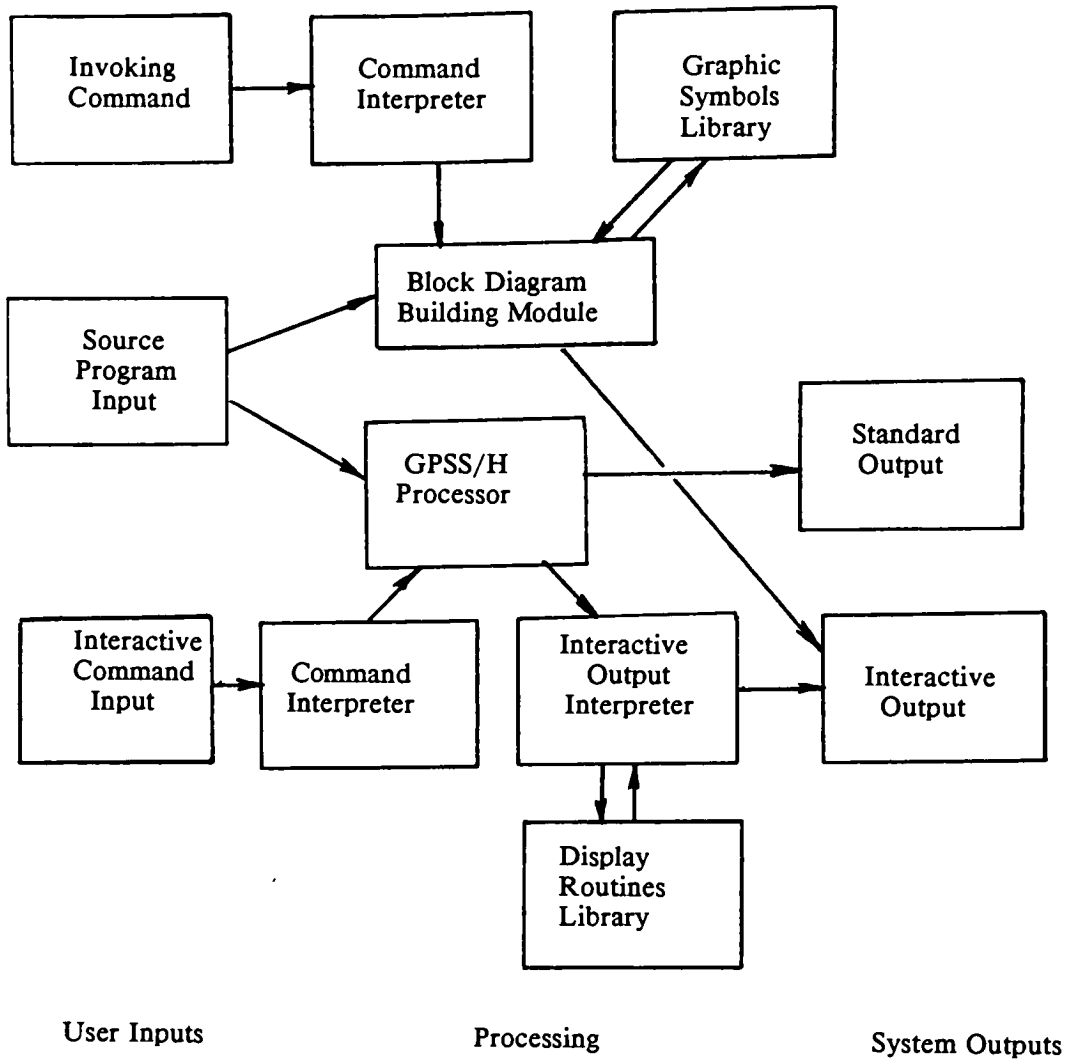


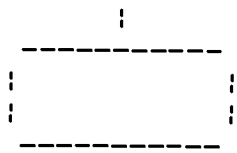
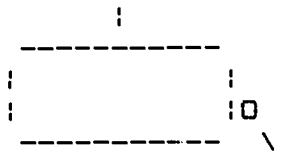
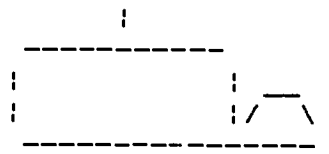
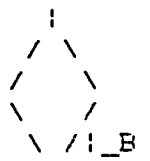
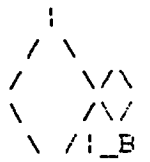
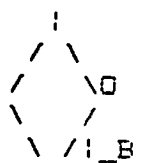
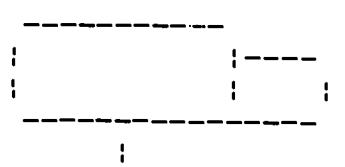
Fig. 4. GPSS/H Visual Debugger I/O architecture.

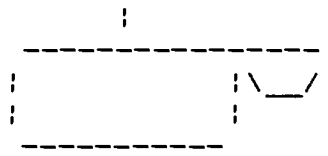
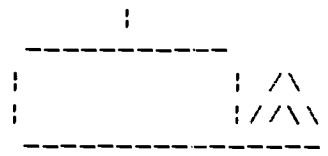
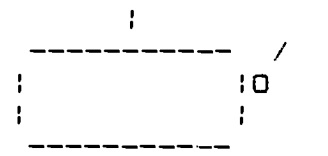
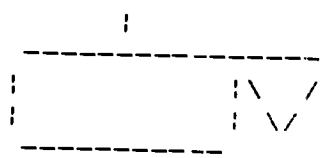
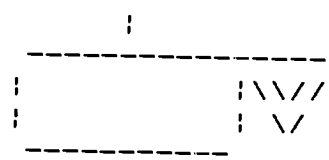
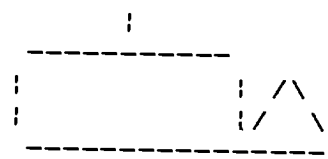
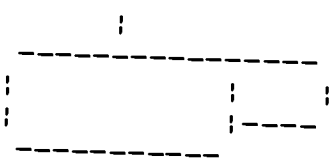
5.3. Graphical GPSS Block Symbols

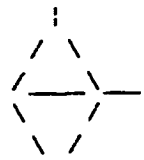
The blocks available in the GPSS/H Visual Debugger are listed in Table 1. For each block are given its name and the symbol suggested by us for display in character mode. For a full description of each block - function, operands, explanation - see Chapter 5 of the GPSS/H User's Manual [9].

Table 1

GRAPHICAL SYMBOLS FOR GPSS BLOCKS

Block Name	Character Mode Block Symbol
ADVANCE	
DEPART	
ENTER	
GATE	
GATE Facility	
GATE Storage	
GENERATE	

Block Name	Character Mode Block Symbol
LEAVE	
PREEMPT	
QUEUE	
RELEASE	
RETURN	
SEIZE	
TERMINATE	

Block Name	Character Mode Block Symbol
TEST	
TRANSFER	

6. GPSS/H Visual Debugger Commands

The following section describes the available commands in the GPSS/H Visual Debugging System and their usage. It can also serve as a simplified user manual.

With the exception of the initiation command, it was decided to use the same conventions as the ones employed in the GPSS/H Reference Manual [9]. These are:

- Minimal abbreviations are indicated by underscoring.
- Optional items are enclosed in brackets (" [ ] ").
- Mandatory items may be enclosed in braces (" { } ").
- When more than one choice exists for an item, choices are shown on the same line separated by " | " characters, or choices are shown on separate lines.
- An ellipsis (" . . . ") is used to denote an optional repetitions of a preceding item.
- Debugger commands can be entered in either upper or lower case characters, or any combination of both.

For example, the command:

BREAK { block } . . .

requires at least one block to be specified (but allows more than 1), and the command can be abbreviated as BREA, BRE, BR or B.

The syntax of the invoking command is kept the same as the syntax of the commands in the host operating system - in this case UNIX.

6.1. Invoking the GPSS/H Visual Debugger

The GPSS/H Visual Debugging System is invoked with the following command:

gpsdbg [options] file[.gps]

file.gps is the name of the file containing the GPSS source code. The suffix *.gps* is required in the file name although the user does not have to enter it in the command line.

options is any number of the following options:

-T ttname	name of the user terminal. The name should be the same as the one used in the */etc/termcap* database. The default is the current value of the $TERM environment variable.
-n n	the starting block number of the Block diagram segment to be displayed. The default value is 1.
-d ns	delay factor for transaction movement display. n is the number of blinks and s is the pause between blinks. The default values are 101 for slow transmission lines (300 baud) and 301 for fast lines (> 9600 baud).
-t on	enable trace mode.
-t off	disable trace mode (default).
-x on	display transaction numbers.
-x off	do not display transaction numbers (default).
-p str	pass the options in the character string *str* to the GPSS command processor.

6.2. Terminating the GPSS/H Visual Debugger

The debugging session can be terminated by issuing any one of the following commands:

STOP

QUIT

QQ

6.3. Debugger Environment Control Commands

The GPSS/H Visual Debugger provides a number of commands that can be used to tailor the Debugger environment according to the user's needs. The user can record a log of the debugging session, enable/disable the execution trace facility, set CPU time limits, save/restore intermediate stages of the model execution, and define command sequences that are to be executed when specified Blocks are reached.

6.3.1. Controlling the Terminal Log (TLOG)

The user can record a log of the debugging session. If TLOG is turned ON, all interactive input and output lines will be recorded in the bulk (standard) output file. In the log, the input lines are prefixed with an arrow. The syntax of the command to control the terminal log feature is:

SET TLOG { ON | OFF }

6.3.2. Controlling the Execution Trace

When executing multiple steps, the user can request the debugger to either display all intermediate steps, or display only the last step. This is relevant for the CONTINUE, NEXT, RUN and STEP n commands. The syntax of the command to control the tracing feature is:

SET TRACE { ON | OFF }

6.3.3. Setting Time Limits

The user can specify CPU execution time limits which supercede any currently active time limits, including limits set by SIMULATE statements within the model. The syntax of the command is:

SET TIME n.n [S | M]

where an *S* suffix indicates that the time limit is specified in seconds, and an *M* suffix (or if no suffix is given) indicates that the time limit is specified in minutes.

6.3.4. Recording Memory Image of the Model

At any given moment, the user can issue the CHECKPOINT command and the Debugger will save a memory image of the model in a disk file. Subsequently, the user can issue the RESTORE command and the status of the model will be restored to that which was saved with the most recent CHECKPOINT command. At most one model image can be saved at any given time. The syntax of the commands is:

CHECKPOINT

RESTORE

6.3.5. Command Procedures (AT-points)

Command procedures (or At-points) in the Debugger environment are sequences of one or more Debugger commands that are to be executed every time a transaction attempts to enter a specified Block or Blocks. The At-points are specified using the AT command which has the following syntax:

```
AT block ...
dbg_command_1
dbg_command_2
   ...
dbg_command_n
END
```

The syntax of the individual commands in a command procedure is checked at execution time. Commands with syntax errors result in error messages and execution continues with the next command. TRAP conditions can not be used in AT commands.

At-points are removed with the UNBREAK command (see Section 6.4.2).

6.4. Model Execution Control Commands

The commands that control the actual execution of GPSS Blocks can be divided into two major subgroups: commands for beginning and resuming of model execution, and commands for suspending of model execution.

6.4.1. Beginning and Resuming Model Execution

Actual execution of GPSS statements is triggered with the STEP, RUN, CONTINUE and NEXT commands.

The STEP command is used to execute a specified number of blocks. If no count is given, 1 is assumed. The syntax of the command is:

STEP [n]

The RUN command is used to initialize normal execution of the model. Optionally, the user can specify one or more global breakpoints and/or trap conditions. The syntax of the RUN command is:

RUN [block | NEXT | SYSTEM | CLOCK value | XACT n]

The CONTINUE command is used to initialize normal execution of the model. Optionally, the user can specify one or more local breakpoints. The syntax of the CONTINUE command is:

CONTINUE [block] ...

The NEXT command is used to continue execution until the simulator picks up the next active transaction from the Current Events Chain. The SYSTEM global trap condition is ignored (if active) for the duration of the NEXT command. The syntax of the NEXT command is:

NEXT

6.4.2. Suspension of Model Execution

Two mechanisms are offered for controlled suspension of program execution: breakpoints and traps.

Local or global breakpoints can be established at any block in the model. Local breakpoints remain in effect only for the duration of a single command. Global breakpoints remain in effect until they are explicitly removed. Local breakpoints can be established with the CONTINUE command (see Section 6.4.1). Global breakpoints can be established with the RUN command (see Section 6.4.1), or with the BREAK command. The syntax of the BREAK command is:

BREAK { block } ...

Associated with each global breakpoint is an ignore count. Each time a breakpoint is encountered, its ignore count is checked and if it is greater than zero, the ignore count is decremented by one and the breakpoint ignored. Thus, a breakpoint is recognized only if its ignore count is zero. Initially, when a breakpoint is established, its ignore count is set to

zero, but the Debugger provides the IGNORE command which can be used to alter that. The syntax of the command is:

IGNORE block count

Global breakpoints can be removed using the UNBREAK command. The syntax of the command is:

UNBREAK block ...

The GPSS/H Debugger provides a mechanism for trapping a number of special conditions. When a trap for a particular condition is enabled and the condition arises, a message is issued and control returns to the user. The following conditions can be specified:

SYSTEM	triggered when a transaction that is moving trough the model can not move further.
NEXT	triggered when the simulator picks up an active transaction from the Current Events Chain.
CLOCK	triggered when the Absolute clock equals or exceeds a specified value.
XACT	triggered every time a specified transaction is picked up as an active from the Current Events Chain.

The syntax of the TRAP command is:

TRAP [NEXT | SYSTEM | CLOCK value | XACT n]

To disable a previously defined trap condition, the user can issue the UNTRAP command. The syntax of the command is:

UNTRAP [NEXT | SYSTEM | CLOCK value | XACT n]

6.5. Output Control Commands

The user can display model status and data with the help of the DISPLAY and PRINT commands. The DISPLAY command displays information on the screen, while the PRINT command records the information in the standard (bulk) file. Both commands can be used to display a wide variety of model status information (breakpoints, storage, clocks, etc.), different entity classes (facilities, queues, logical switches, etc.), individual ampervariables, transactions, etc.

For a full description of the different entities that can be displayed with the DISPLAY and PRINT commands see the GPSS/H User's Manual [9].

6.6. Help Command

The GPSS/H Visual Debugger provides on-line help facility. When the user issues the HELP command, the Debugger will display on the screen a list of the most commonly used commands together with a brief explanation of their use. The syntax of the HELP command is:

HELP

6.7. Summary of the GPSS/H Visual Debugger Commands

The Debugger commands described in the previous sections are summarized in Table 2.

Table 2

Command	Operands	Explanation
AT	One or more Block names or numbers	The AT command prompts the user for a list of interactive debugging commands to be executed every time a Transaction reaches any of the designated Blocks. The list of commands is terminated by typing END. AT-points remain in effect until explicitly removed with the UNBREAK command.
BREAK	One or more Block names or numbers	The BREAK command sets global breakpoints at each of the specified Blocks. Global breakpoints remain in effect until removed with the UNBREAK command. A global breakpoint can be temporarily suppressed by using the IGNORE command.
CHECKPOINT	none	The CHECKPOINT command writes into the checkpoint file a memory image of the model being debugged.
CONTINUE	Zero or more Block names or numbers	The CONTINUE command is used to resume execution of a model. Any Block names or numbers specified in the CONTINUE command are set as local breakpoints, which remain in effect only for the duration of the CONTINUE command.
DISPLAY	-->	The DISPLAY command can select a wide variety of model statistics and other useful information for display on the terminal. The user is limited to one statistic per call.
HELP	none	The HELP command results in the display on the screen of a list with the most frequently used debugger commands and a brief explanation of their usage.
IGNORE	A Block name or number, followed by a count	The IGNORE command is used to temporarily suppress a global breakpoint. The count specifies the number of times the breakpoint is to be ignored.
NEXT	none	The NEXT command specifies that execution of the model is to proceed until the next active Transaction is picked from the Current Events Chain. If trap-conditions (with the exception of the SYSTEM trap), breakpoints, or at-points are encountered by the current Transaction, the NEXT command is aborted.

Table 2 (contd.)

Command	Operands	Explanation
PRINT	-->	The PRINT command can write a wide variety of model statistics and other useful information into the standard output file or device (not the terminal).
QQ	none	The QQ command causes immediate termination of a run. No check is made for pending output.
QUIT	none	The QUIT command is used to terminate a run. If any model output is pending at the time the QUIT command is issued, a warning is given and the command must be typed a second time to suppress the output.
RUN	Zero or more Block names or numbers	The RUN command is used to initiate execution of a model. Global breakpoints are set for any Blocks designated on the RUN command.
	SYSTEM NEXT CLOCK [=] n XACT [=] n	Global trap-conditions can be set by the RUN command. For a summary of these conditions, see the description of the TRAP command.
RESTORE	none	The RESTORE command is used to restore model status to that which was previously written into the checkpoint file by means of a CHECKPOINT command.
STEP	optional count	The STEP command is used to step through a specified number of Block executions. If no count is given, a value of one is assumed.
STOP	none	The STOP command is used to terminate a run. If any model output is pending at the time the STOP command is issued, a warning is given and the STOP command must be be typed a second time to suppress the output.
SET	TIME [=] n TIME [=] nS TIME [=] nM	The SET command can be used to set a CPU time limit which supersedes any currently active time limit, including time limits set by SIMULATE statements within a model. An *S* suffix indicates that the time limit is specified in seconds. An *M* suffix, or no suffix indicates that the time limit is specified in minutes. Non-integral values (e.g. SET TIME=3.5M) are not allowed.

Table 2 (contd.)

Command	Operands	Explanation
	TLOG [=] ON TLOG [=] OFF	The SET command can be used to turn the terminal log on and off. When the terminal log is turned on, all interactive debugging input and output are also written into the standard output file or device. Input lines (commands) are prefixed by an arrow.
	TRACE [=] ON TRACE [=] OFF	The SET command can be used to turn the trace facility on and off. When TRACE is on, all intermediate steps of the RUN, CONTINUE and STEP *n* commands are displayed on the screen.
TRAP	SYSTEM	The TRAP SYSTEM command specifies that each time the active Transaction comes to a rest, a message is to be issued, and control is to return to the user.
	NEXT	The TRAP NEXT command specifies that each time a new active Transaction is picked up in the simulator's scan of the Current Events Chain, a message is to be issued, and control is to return to the user.
	CLOCK [=] n	The TRAP CLOCK [=] *n* command specifies that when the Absolute Clock reaches or exceeds the specified value, a message is to be issued, and control is to return to the user. At most one clock trap-condition can be in effect at a time.
	XACT [=] n	The TRAP XACT [=] *n* command specifies that a Transaction number *n* is to be flagged in such a manner that each time it is picked up as an active Transaction in the simulator's scan of the Current Events Chain, a message is to be issued, and control is to return to the user.
UNBREAK	One or more Block names or numbers	The UNBREAK command is used to remove global breakpoints and at-points.
UNTRAP	SYSTEM NEXT CLOCK [=] n XACT [=] n	The UNTRAP command removes traps set via the TRAP command.

7. HIGH LEVEL MODULE DESIGN

The main flow of control in the GPSS/H Visual Debugging System is depicted on Figure 5. There are two distinctive stages, corresponding to the "static" and "dynamic" parts of the debugging session - see Section 3.1.

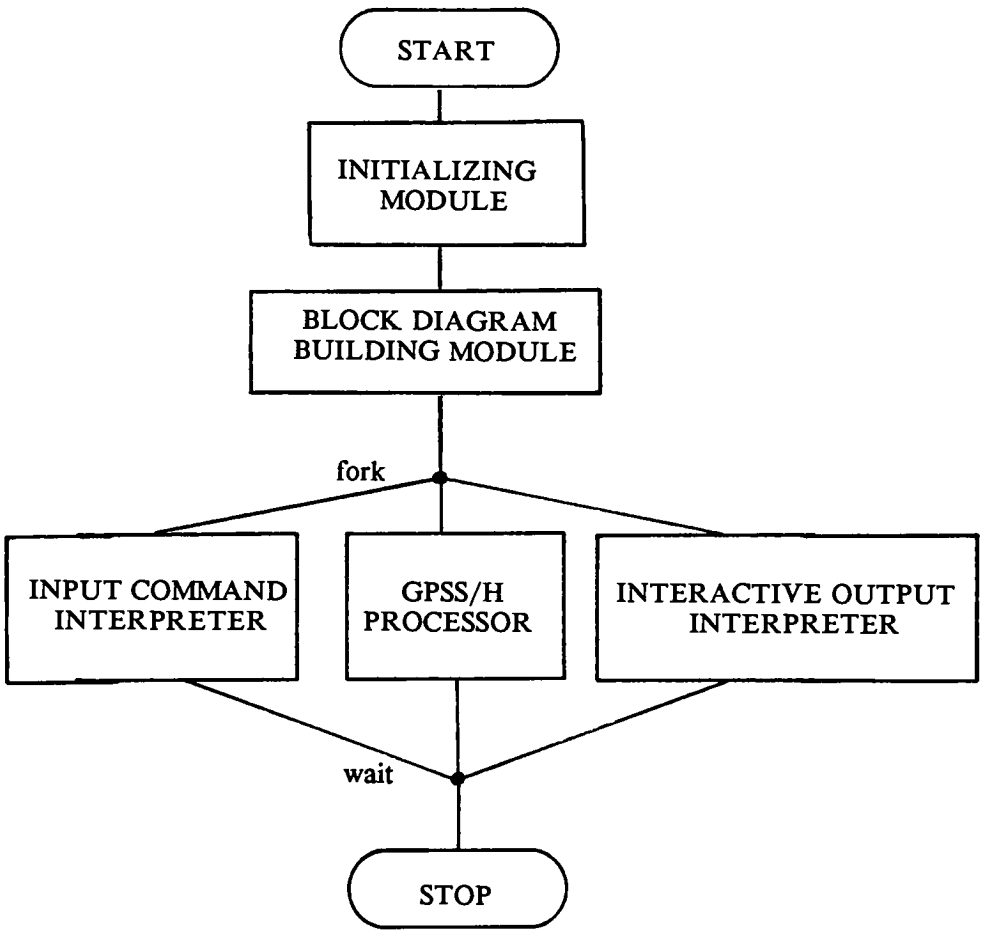


Fig. 5. GPSS/H Visual Debugger - Main flow of control.

During the static stage, the flow of control has the classical sequential execution nature. The last step in the static stage is the creation of the three separate subprocesses that comprise the dynamic part of the system. These subprocesses execute in parallel and are synchronized via a set of pipes - see Section 7.5.

7.1. Initialization Module

The Initialization module contains the entry point and controls the activities of the whole system during the static part of the debugging session (see Section 3.1). It performs the following functions:

(1) Initialize all optional parameters to binary zero.

(2) Parse the initiation command and separate the options from the file name.

(3) Check whether the specified file name ends in *.gps*. If not, append *.gps* to the file name.

(4) Check whether the specified file exists. If not, print an error message and exit with return code of 1.

(5) Check the validity of the requested options. If an option is unknown, issue a warning message and continue with the next one. If the option is legal but the requested value is invalid, assign the default value for that particular option and issue a warning message to this effect. Repeat the above until all options are processed.

(6) If warnings were issued, ask the user whether to continue or not. If the answer is negative, exit with return code of 2.

(7) Scan the internal list of optional parameters. Assign the appropriate default values to every parameter that is still equal to binary zero (has not received a value as a result of step 5).

(8) Open the interprocess communication pipes. If not successful, issue a system error message and exit with a return code of -1.

(9) Initialize the screen map structure.

(10) Initialize the screen in the appropriate graphics mode. If not successful, issue a system error message and exit with a return code of -2.

(11) Build the requested section of the Block diagram. If syntax errors are encountered, issue an error message and exit with a return code of 3.

(12) Create and start the Interactive Command Interpreter (see Section 7.3). If not successful, issue a system error message and exit with a return code of -3.

(13) Create and start the GPSS/H subprocess. If not successful, issue a system error message and exit with a return code of -4.

(14) Create and start the Interactive Output Interpreter (see Section 7.4). If not successful, issue a system error message and exit with a return code of -5.

(15) Close all unneeded file descriptors.

(16) Wait for any of the subprocesses to exit. Normally, the GPSS/H subprocess should be the one to exit first since the other two subprocesses are in effect infinite loops.

(17) Issue an INTERRUPT to kill the remaining subprocesses.

(18) Restore the initial screen environment and exit with a return code of zero.

7.2. Block Diagram Building Module

The primary task of the Block Diagram Building Module is the drawing on the screen of the requested (or the default) section of the Block diagram and the creation of a symbol table (map) with all information that might be later needed by the other modules.

The Block Diagram Building module is invoked by the Initialization module which passes the name of the GPSS source file as a parameter. The module performs the following functions:

(1) Open the source file in read mode. If not successful, issue an error message and send a kill signal to all subprocesses.

(2) Read the source file line by line. For each line do the following:

  (2.1) Initialize the appropriate variables.

  (2.2) Determine whether this is an empty or a comment line. If so, skip to the next line.

  (2.3) Get the label (if present) and the GPSS command.

  (2.4) Check whether the GPSS command is a valid block statement. If not, skip to the next line.

(2.5) Increment the block count and check whether the block is implemented. If not, skip to the next line.

(2.6) Get the parameter list.

(2.7) Record all relevant information in the symbol table.

(2.8) Put the symbol for the block in the screen image.

(2.9) Adjust the pointers to the symbol table.

(3) Add the headings of the standard fields to the screen image.

(4) Display the screen.

(5) Close the source file

(6) Return to the Initialization module

7.3. Input Command Interpreter

The Input Command Interpreter passes the user commands to the GPSS processor. It runs as a separate subprocess and communicates with other modules (i.e., the GPSS processor and the Interactive Output Interpreter) via a set of pipes; see Section 7.5. The Input Command Interpreter is in effect an infinite loop which performs the following functions:

(1) Read the user command from the user terminal.

(2) Parse the command line into command and parameters.

(3) Map all characters of the command into lower case characters.

(4) Determine whether the command is valid (allowing for legal abbreviations).

(5) If the command is valid, call the appropriate routine to handle the request.

(6) If the command is not valid, ignore it and issue a message.

7.4. Interactive Output Interpreter

The Interactive Output Interpreter module is similar to the Input Command Interpreter module. It also runs as a separate subprocess and communicates with the other modules via a set of pipes; see Section 7.5. First, the process is initialized by reading the "Ready!" line of the Interactive GPSS output. If the read fails, a system error message is issued and a kill signal is sent to all subprocesses. Otherwise, the module enters an infinite loop for the remaining of the session and the following functions are performed:

(1) Clear the screen from any leftover information from previous requests.

(2) Read a line from the GPSS Interactive output.

(3) Send the appropriate continuation signal to the Interactive Command Interpreter.

(4) Compare the line just read to the set of expected lines.

  (4.1) If it is an empty line, display the Visual Debugger prompt.

  (4.2) If it is a DISPLAY command header line, display on the screen the corresponding Visual Debugger header information.

  (4.3) If it is a DISPLAY command data line, display on the screen the appropriate data.

  (4.4) If it is a transaction movement related line, get the variable information, determine whether the block is displayed on the screen and if so, perform the appropriate action. Update the relative clock display.

  (4.5) If it is a HELP data line, display the HELP information.

  (4.6) If it is a SIMULATION TERMINATED line, display the "how to exit" message.

(5) If there is no match, ignore the line and continue with the next one.

7.5. Interprocess Communication

The synchronization of the different subprocesses - Input Command Interpreter, GPSS/H processor and Interactive Output Interpreter - is achieved via a set of unnamed pipes [2]. The diagram on Figure 6 shows schematically the three subprocesses and the redirection of their standard inputs and outputs.

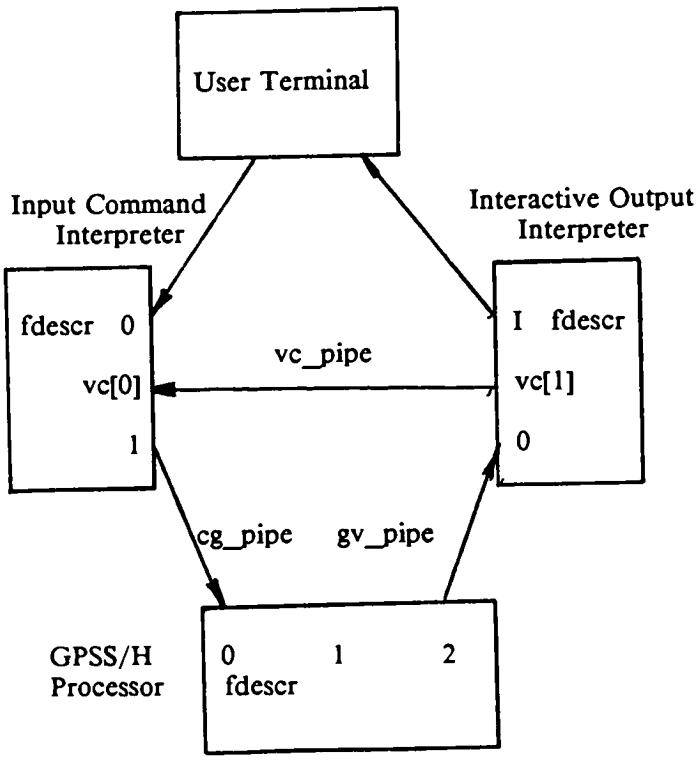


Fig. 6. Interprocess Communication

7.6. Maintaining the Screen Image

The following discussion is based on the *curses* library package [1]. There were two major reasons for selecting the *curses* package for this implementation: first, it is a standard part of the UNIX environment, and second, it is capable of displaying and reading information from a large number of terminal devices which are described in the */etc/termcap* database.

Basically, the *curses* package works in the following way:

(1) The environment is initialized and information about the terminal is pulled from the terminal capabilities database and made available to the *curses* routines.

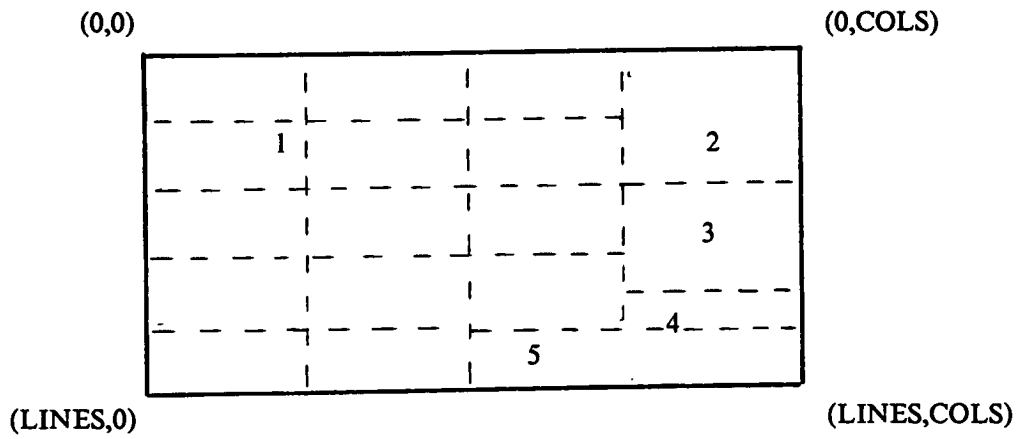
(2) An image of the desired screen is built in memory.

(3) The information is displayed on the actual screen (by making a call to the *refresh( )* routine).

(4) Steps (2) and (3) are repeated as many times as needed.

(5) At the end of the program the original environment is restored.

In the case of the GPSS/H Visual Debugging System the screen is logically divided as shown on Figure 7.



where 1 - Block diagram symbol area
2 - Relative clock area
3 - Help display area
4 - Interactive command input area
5 - Text output area

Fig. 7. Logical screen layout.

In line with the *curses* package, the coordinate of the upper left corner are assumed to be (0,0) and the coordinates of the lower right corner are (LINES,COLS). The values for LINES and COLS are pulled from */etc/termcap*. All dimensions are calculated from these two values with the help of several constants (defined in a separate *.h* file).

During initialization time the centers of the Block diagram areas are recorded in structure *screen* which plays the role of a symbol table and has the following declaration:

```
struct screen {
        int x_coord;
        int y_coord;
        int block_No;
        int block_code;
        char *label;
        char *block;
        char *parameters;
};
```

In this way it becomes possible to use relative addressing for all symbol drawing and other block related display routines.

The other areas (command input, help, etc.) are addressed relative to the coordinates of the lower right corner (LINES,COLS).

8. CONCLUSIONS

The presented system is an effort to create a debugger for the GPSS language that will further ease the difficult and time consuming process of producing an error free computer simulation models of different real life systems. The proposed debugging system introduces several new features while preserving virtually all of the capabilities of the existing debugger. The presented version of the GPSS/H Visual Debugger offers a wide range of features, more notable of which are:

- Interactive processing
- Full-screen display
- Execution trace facility
- Symbolic referencing of entities
- Capabilities for stepping through the program
- Capabilities for imbedding break and trace points
- Selective display of model statistics
- Capabilities for customizing
- On-line help facility

On the other hand, the current version of the GPSS/H Visual Debugger has some key weaknesses.

- Limited number of Blocks displayed on the screen. (15 for the standard 24x80 terminals.)
- Inability to deposit/modify data during a debugging session.
- Lack of Operating System interface (exists only in line mode and not for all host operating systems).
- Inability to evaluate arithmetic expressions using symbolic and absolute data.

- Inability to display the source code upon request.
- Lack of a mechanism for creating a user defined commands.

The above remarks are based on the experience gained from using the GPSS/H Debugger for the debugging of several models and the author's study of other debugging systems.

Despite its weaknesses, the GPSS/H Visual Debugger is a versatile and powerful tool. Like any other complex software product, it requires some time and a certain level of programming experience to learn how to use it. At the same time, in many respects, the Visual Debugger is easier to learn than other debuggers. The Debugger offers the experienced programmer a wide range of capabilities which may seem like "overkill". As is the case with other similar products, each user would probably become accustomed to only a subset of the commands.

A determined effort has been made to design the system in such a way as to allow easy modification and upgrading. For example, all graphics related routines use relative coordinate addressing and generic type requests. This will allow the relatively easy implementation of new types of terminals or graphics packages - all that would be necessary to change would be a couple of *.h* files and the addition of new *when* clauses in the appropriate *switch* statements.

The greatest problem that was encountered during the development of the Visual Debugger was the lack of access to the source code and internals of the actual GPSS/H processor.* This frequently forced "not so clean" solutions for certain parts of the system. This is especially true for the output end of the system where the user oriented interactive output in line mode is captured, so that the necessary data can be extracted and passed to the appropriate screen display routines. The lack of access to the code of the GPSS/H processor

---

*) GPSS/H is a proprietary product of the Wolverine Software Corporation.

also led to the decision to divide the system into three more or less independently running subprocesses which in turn poses many problems with module synchronization and communication.

The above mentioned drawbacks are the most logical places for future improvement of the system. Other possible improvements and extensions of the GPSS/H Visual Debugger could be:

- The creation of scrollable screens which will permit the display of larger portions of the model.
- The gradual implementation of different types of terminals operating in graphics mode.
- The implementation of a mechanism for switching from screen mode to line mode and vice versa in the middle of a session.
- The creation of an operating system interface that will allow the execution of operating system commands during the debugging session.
- The implementation of capabilities for keyboard function keys definition. This will allow the user to predefine complex command sequences or frequently used commands and execute them by striking just one key.

The performance characteristics of the Debugger have not been a major consideration in the current version. Keeping in mind that debugging is a slow process to begin with, in terms of user response time, performance should not be an issue.

9. BIBLIOGRAPHY

[1] K. Arnold. "Screen Updating and Cursor Movement Optimization." University of California, Berkeley, 1980.

[2] M. J. Bach. The Design of the UNIX Operating System. Prentice-Hall, Inc, 1986.

[3] P.A. Bobillier, B.C. Kahan, A.R. Probst. Simulation with GPSS and GPSS V. Prentice-Hall, Inc, 1976.

[4] J.D. Foley, A. Van Dam. Fundamentals of Interactive Computer Graphics. Addison-Wesley Publishing Company, 1983.

[5] Jacquelyn Van Dellon. "GPSS Compiler/Simulator." MS Thesis, Rochester Institute of Technology, 1984.

[6] G. Gordon. "A General Purpose Systems Simulator." IBM Systems Journal, September, 1962.

[7] G. Gordon, R. Efron. "A General Purpose Digital Simulator and Examples of its Use: Part 1 - Description of the Simulator." IBM Systems Journal, Vol. 3, No. 1, 1964.

[8] P. Gubitosa. "CICS-VS: An On-line Software Monitor." Van Nostrand Reinhold Company, 1984.

[9] J. O. Henriksen, R. C. Crain. GPSS/H User's Manual. Wolverine Software Corporation, 1983.

[10] H. Herscovitch, T. Schneider. "GPSS III - An Expanded General Purpose Simulator" IBM Systems Journal, Vol. 4, No. 3, 1965.

[11] B.W. Kernighan, R. Pike. The UNIX Programming Environment. Prentice-Hall, Inc, 1984.

[12] B.W. Kernighan, D.M. Ritchie. The C Programming Language. Prentice-Hall, Inc, 1978.

[13] J.F. Maranzano, S.R. Bourne. "A Tutorial Introduction to ADB." Bell Laboratories, 1977

[14] H.D. Mills. "Principles of Software Engineering." IBM Systems Journal, Vol. 19, No. 4, 1980.

[15] R.E. Quinan. "Software Engineering Management Practices." IBM Systems Journal, Vol. 19, No. 4, 1980.

[16] C.B. Richards. GPSS/C Quick Reference Manual. London, Ontario, Canada, 1982.

[17] VAX/VMS Symbolic Debugger Reference Manual. Digital Equipment Corporation, 1986.

[18] Virtual Machine/System Product 5 CP Command Reference for General Users. SC24-2576, IBM Corporation, 1986.

APPENDIX A: Sample GPSS Model

The following section is intended for people with limited knowledge of computer simulation with the GPSS language. The goal is to give the reader a general idea of the major stages through which one passes when developing a simulation project.

First, the problem has to be stated. Let us say, for example, that we have to simulated the activities in a small computer laboratory which has only one terminal. We are interested in determining how well is this terminal utilized and how much time is waisted by the students waiting to get access to that terminal. During the stage of problem formulation, a decision has to be made on the level of abstraction and what are the things that have to be included in the model and what can be ignored. For example, in our case we decided that we are not interested in what are the students doing while waiting, are they waiting at all or are they signing on a waiting list and coming later, etc.

The second stage would be to collect system data. Usually this is done through long term observations of the behavior of the real life system and the use of classical statistical tools (regression analysis of variance, etc.) to convert the recorded data into form suitable for the simulation study. Let us assume that for our example we have determined the interarrival time of the students to be 32 ± 10 min. uniformly distributed, the duration of a terminal session to be 29 ± 8 min. (also uniformly distributed), and that the lab is open 10 hours a day (600 min.).

The next stage of the modeling process would be the creation of a GPSS Block diagram that describes the system being modeled in terms of the basic GPSS building blocks.

Once the Block diagram is created, we are ready to code the actual GPSS source program. The source code for our example is shown on Figure 8.

```
          SIMULATE
*
*         Users coming to the lab
*
          GENERATE 32,10          ; people arrive every 32 +- 10 min.
          QUEUE    DOOR           ; wait in line for the terminal
          SEIZE    TTY1           ; log-on the system
          DEPART DOOR             ; leave the queue
          ADVANCE 29,8            ; do your work
          RELEASE TTY1            ; log-off
          TERMINATE


*
*         Simulated Clock
*
          GENERATE 600,,,1        ; the lab is open 10 hrs.

          TERMINATE 1
          START     1

          END
```

Fig. 8. GPSS source code for the computer lab model.

After the model is executed, the GPSS processor produces a statistical report - see Figure 9. The report gives a variety of statistical information about the status of the model at the end of the simulation, on the utilization of the different facilities (in our case the terminal), on the average time a transaction (a student) spent in the queue, etc.

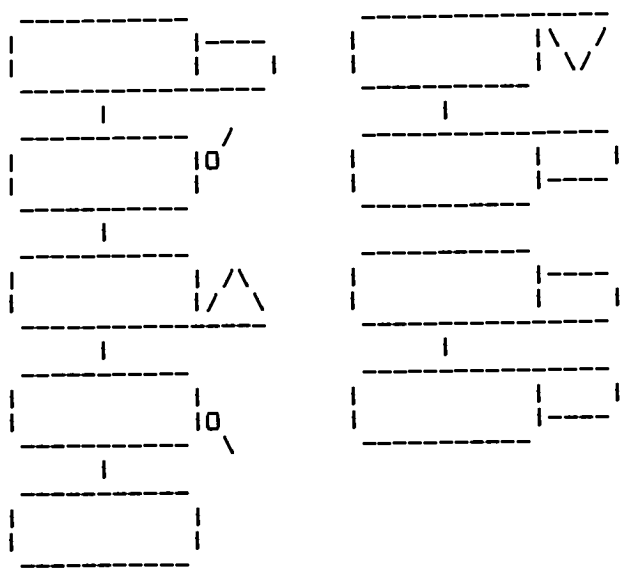
The final stage of a simulation project would be to statistically verify the results of the simulation and prepare an appropriate report.

```
GPSS/H VAX/UNIX RELEASE 0.93 (UL195)     20 Jul 1987   17:27:15     FILE: wq.gps

LINE# STMT# BLOCK#  *LOC   OPERATION   A,B,C,D,E,F,G   COMMENTS

    1     1                 SIMULATE
    2     2          *
    3     3          *      Users coming to the lab
    4     4          *
    5     5       1         GENERATE 32,10       ; people arrive every 32 +- 10 min.
    6     6       2         QUEUE    DOOR        ; wait in line for the terminal
    7     7       3         SEIZE    TTY1        ; log-on the system
    8     8       4         DEPART DOOR          ; leave the queue
    9     9       5         ADVANCE 29,8         ; do your work
   10    10       6         RELEASE TTY1         ; log-off
   11    11       7         TERMINATE
   12    12          *
   13    13          *      Simulated Clock
   14    14          *
   15    15       8         GENERATE 600,,,1     ; the lab is open 10 hrs.
   16    16       9         TERMINATE 1
   17    17                 START     1
   18    18                 END

ENTITY DICTIONARY (IN ASCENDING ORDER BY ENTITY NUMBER; "*" => VALUE CONFLICT.)

      Facilities: 1=TTY1
          Queues: 1=DOOR

SYMBOL   VALUE   EQU DEFNS  CONTEXT      REFERENCES BY STATEMENT NUMBER

TTY1         1              Facility        7    10
DOOR         1              Queue           6     8

Simulation begins.

RELATIVE CLOCK: 600.0    ABSOLUTE CLOCK: 600.0

BLOCK CURRENT     TOTAL      BLOCK CURRENT     TOTAL      BLOCK CURRENT     TOTAL
1                    17      4                    17      7                    17
2                    17      5                    17      8                     1
3                    17      6                    17      9                     1


         --AVG-UTIL-DURING--
FACILITY  TOTAL  AVAIL  UNAVL     ENTRIES    AVERAGE   CURRENT  PERCENT  SEIZING
           TIME   TIME   TIME               TIME/XACT  STATUS   AVAIL     XACT
    TTY1  0.833                        17      29.408   AVAIL

  QUEUE      MAXIMUM   AVERAGE    TOTAL     ZERO   PERCENT    AVERAGE    $AVERAGE
             CONTENTS  CONTENTS   ENTRIES   ENTRIES  ZEROS   TIME/UNIT  TIME/UNIT
   DOOR             1     0.066        17         8   47.1       2.346      4.432


Simulation terminated.  Absolute Clock:     600.00
```

Fig. 9. GPSS listing.

APPENDIX B: Sample Screen Layout

The following figure shows a sample screen layout. It shows the Block diagram of the computer lab simulation problem (see Appendix A) and the Visual Debugger help display.



```
REL.CLOCK:
 109.1552

quit
   Exit
qq
   Exit
c
   Continue
s [n]
   Step n
d xxx
   Display
?
   Help

dbg››› help
```

Fig. 10. Sample screen layout.

APPENDIX C: List of Unsupported Blocks

ALTER
ASSEMBLE
ASSIGN
BCALL
BCLEAR
BGETLIST
BGETSTRING
BLET
BPUTPIC
BPUTSTRING
BRESET
BRMULT
BSTORAGE
BUFFER
CHANGE
COUNT
EXAMINE
EXECUTE
FAVAIL
FUNAVAIL
GATHER
HELP
INDEX
JOIN
LINK
LOGIC
LOOP
MARK
MATCH
MSAVEVALUE
PRINT
PRIORITY
REMOVE
SAVAIL
SAVEVALUE
SCAN
SELECT
SPLIT
SUNAVAIL
TABULATE
TRACE
UNLINK
UNTRACE
WRITE